실험 8. Multi-Tasking 동기화-2

1. 목적

이 실험은 실시간 운영체제가 제공하는 멀티태스킹 환경에서, 간단한 애니메이션이 있는 두 태스크 사이의 동기화 문제를 해결하는 실험이다.

2. 실험 내용 설명

이 실험에서는 세마포를 이용한 동기화 기법을 이용하여 독립적인 두 개의 태스크가 서로의 영역을 침범하지 않고 화면에 그림을 그리는 프로그램을 작성한다.

이런 세마포를 이용하기 위해서는 RTOS가 지원하는 Semaphore Primitive를 이용할 필요가 있다. http://www.freertos.org/의 <API Reference> - <Semaphore/Mutexes> 섹션을 참고하면 된다. 앞 홈페이지에는 각 API에 대한 설명과 예제가 나와 있다.

주요 사용 API는 vSemaphoreCreateBinary(), xSemaphoreTake(), xSemaphoreGive() 이다.

이 함수들을 이용하려면 "semphr.h"를 #include 해야 한다.

(1) **vSemaphoreCreateBinary**(xSemaphoreHandle xSemaphore)

Macro that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

A binary semaphore need not be given back once obtained, so task synchronization can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. (즉 생성한 뒤, 최초의 Take가 성공한다)

* 실제 vSemaphoreCreateBInary()는 매크로로 다음과 같이 정의된다.

```
#define vSemaphoreCreateBinary( xSemaphore ){\
    xSemaphore = xQueueCreate( ( unsigned portBASE_TYPE ) 1,
                               semSEMAPHORE_QUEUE_ITEM_LENGTH );\
    if( xSemaphore != NULL ){\
        xSemaphoreGive( xSemaphore );\
    }\
}
```

즉, Queue를 하나 할당 받고 (큐 길이는 1, semSEMAPHORE_QUEUE_ITEM_LENGTH - 큐 안의 데이터 길이는 0) xSemaphoreGive를 호출하여 세마포 하나를 바로 take 가능하도록 초기화 된다. 이후에 Queue에 데이터를 꺼내는 동작과 데이터를 넣는 동작으로 Semaphore 기능을 구현하고 있다.

A Binary semaphore is assigned to variables of type **xSemaphoreHandle** and can be used in any API function that takes a parameter of this type.

Parameters:
xSemaphore Handle to the created semaphore. Should be of type xSemaphoreHandle.

(2) **xSemaphoreTake** (xSemaphoreHandle xSemaphore,
portTickType xBlockTime
)

Macro to obtain a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary().

Parameters:
xSemaphore A handle to the semaphore being taken - obtained when the semaphore was created.
xBlockTime The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

Returns:
pdTRUE if the semaphore was obtained. **pdFALSE** if xBlockTime expired without the semaphore becoming available.

(3) **xSemaphoreGive** (xSemaphoreHandle xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary().

Parameters:
xSemaphore A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns:

pdTRUE if the semaphore was released. **pdFALSE** if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

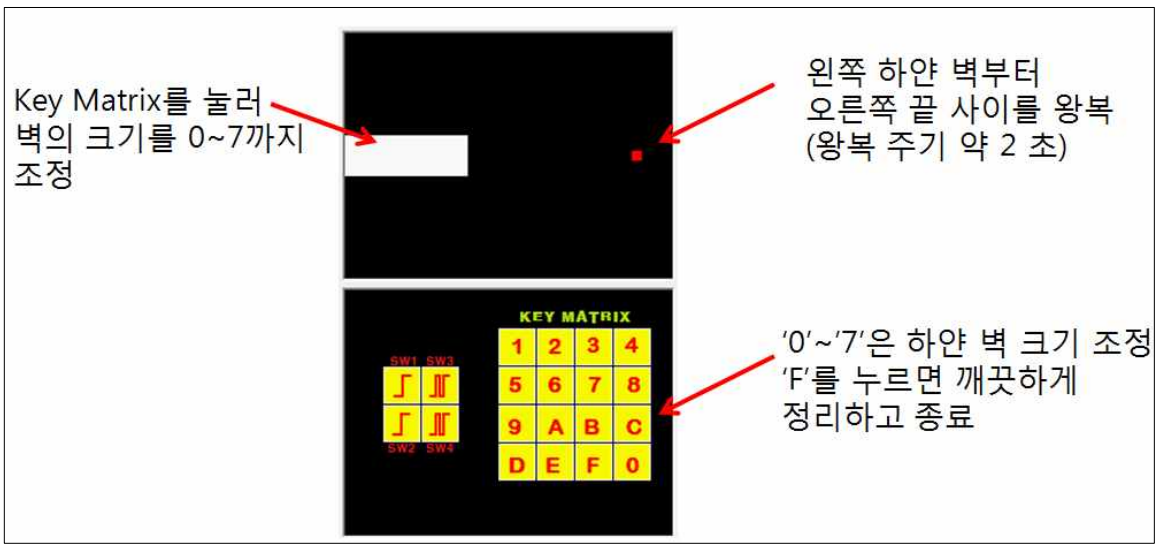
실제 세마포의 사용은 세마포를 생성하고, 두 태스크 간에 Give -> Take 또는 Take -> Give를 주고 받으면서, Event를 알려 실행하는 순서를 제어(동기화)하게 된다. (Give, Take는 우리가 사용하는 FreeRTOS에서 사용하는 용어이며, 이는 up/down, wait/signal, p/v 등 다양한 방법으로 이름이 붙여진다.

3. 예제 소스

이 실험에서는 이전 실험의 box 그리기, Queue를 이용한 KeyMatrix 읽기 등을 모두 이용한다.

4. 응용 실험 (과제)

과제는 두 단계 이다. 첫 단계 (Homework_A)는 닌텐도 DS의 상단 LCD에 하얀 큰 박스(두 번째 단계에서 벽으로 사용)를 키 매트릭스의 입력에 해당되는 위치에 그리는 것이고 두 번째 단계는 LCD 중간에 빨간 볼(작은 8x8 박스)을 **왕복 거리와 상관없이 2초에 한 번씩** 좌우로 왕복시키는 Ball_Task를 만들고, 볼이 움직이는 영역(즉, 벽의 크기)을 Key Matrix에서 입력해 조정해보는 실험이다. 첫 번째 단계는 START Key로 끝낸다.



첫 단계에서는 하얀 Box를 그리는데, 박스의 크기는 32x32 pixel이며, 우리 LCD 크기(266x192)를 고려하면 하얀 박스는 8x6개를 화면에 그릴 수 있다. 큰 하얀 박스 그리는 함수는 작은 box 그리는 함수를 수정해서 작성한다.

실험은 중간 위치의 한 줄을 정해서 하얀 큰 박스와 작은 빨간 박스를 그린다.

첫 단계에서 Start Key를 누르면, 두 번째 단계로 넘어간다. 이 실험의 가장 중요한 부분은 두 번째 단계로 빨간 Ball이 좌우로 왔다 갔다 하는 동안 벽의 크기를 조정할 때, Ball의 위치에 따라 하얀 벽을 그리는 동작과 겹쳐서 벽 위에 Ball을 그리는 일이 벌어지는 것을 방지하는 것이다. 즉, 빨간 Ball을 움직이는 Task와 벽을 움직이는 Task 사이의 동기화를 고려하는 것이다.

실제 이런 동기화 제어 프로그램은 다양한 방법으로 구현될 수 있으나 이번 실험에서는 반드시 Semaphore를 사용하도록 한다.

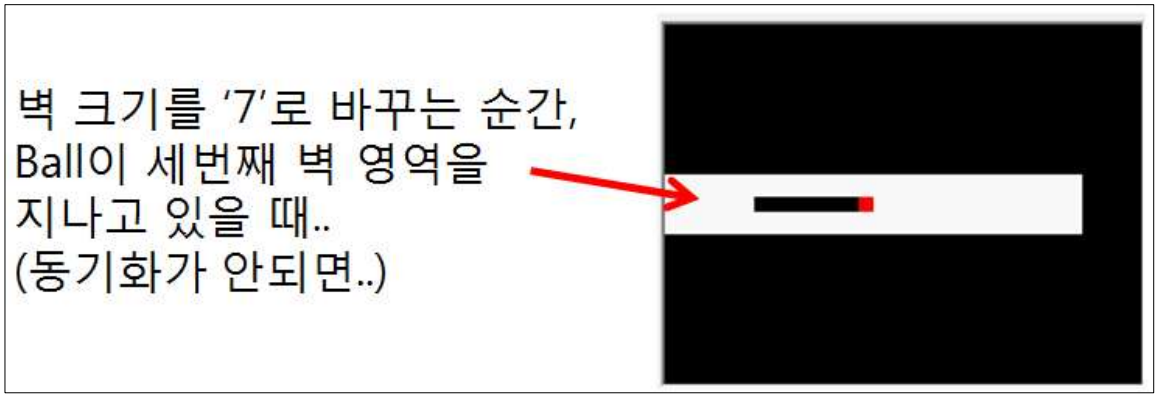
아래는 main.c의 일부로, Ball_Task가 선언되어 있으며, 지난 KeyMatrix 실험 결과인 kbhit(), getkey()등 API를 이용하기 위해 Key_Task도 선언된다. 예제에서 Key Task가 가장 높은 우선순위 (tskIDLE_PRIORITY + 10), Ball_Task는 그 다음 우선순위로, 또 Key를 읽어 Ball이 들어오면 안 되는 하얀 블록 (벽)의 크기를 지정하는 Exp_Task는 Ball_Task 보다도 낮은 가장 낮은 우선순위로 설정한다. 이 우선 순위는 바꾸지 않는다.

```
32    xTaskCreate(Key_Task,
33                        (const signed char * const)"Key_Task",
34                        2048,
35                        (void *)NULL,
36                        tskIDLE_PRIORITY + 10,
37                        NULL);
38
39    xTaskCreate(Ball_Task,
40                        (const signed char * const)"Ball_Task",
41                        2048,
42                        (void *)NULL,
43                        tskIDLE_PRIORITY + 5,
44                        &BallTask);
45    vTaskSuspend(BallTask);
46
47    xTaskCreate(Exp_8_Task,
48                        (const signed char * const)"Exp_8_Task",
49                        2048,
50                        (void *)NULL,
51                        tskIDLE_PRIORITY + 1,
52                        NULL);
53
54    KeyQueue = xQueueCreate(MAX_KEY_LOG, sizeof(u8));
55    // Error Processing Needed !
56
57    vTaskStartScheduler();      // Never returns
```

소스의 라인 45는 Ball_Task가 선언된 뒤에, 다른 Task들과 함께 바로 실행되지 않도록 만든다. Ball_Task는 빨간 Ball을 좌우로 움직이는 Task로 화면에 하얀 박스(벽)를 그리는 첫 번째 단계를 구현한 프로그램 함수, Exp_8_Homework_A()의 실행이 끝난 뒤에 Task를 Suspended 상태로 바꾸는 것이다. (이렇게 하지 않으면, Exp_8_Homework_A()가 실행되는 동안에도 Ball을 그리게 된다.) Exp_8_Homework_A()가 끝난 후, Exp_8_Homework_B()가 시작되면, Ball_Task가 실행하기 위한 준비가 된 후 (세마포 초기화 등), vTaskResume() 함수를 실행하여 그 때부터 Ball_Task가 Ball을 LCD 상에서 왕복하도록 한다. (Exp_8_Homework_B() - template 소스 참조)

실험의 main Task는 Exp_8_Task로 Exp_8_Homework_A()과 Exp_8_Homework_B() 두 함수를 부른다. Exp_8_Homework_A()는 StartKey를 확인해야 하므로, kbhit()와 vTaskDelay() 함수로, Exp_8_Homework_B()는 키보드 입력을 getkey()로 기다리며 (또 Ball_Task와의 동기화를 위한) 대기를 한다.

이 실험에서의 동기화 문제는, 다음 그림과 같이 벽의 크기를 늘릴 때, Ball이 현재로서는 벽이 없는 영역을 지나고 있다면, 벽을 그리는 Task가 벽을 먼저 그린 뒤, Ball이 그 위를 지나가는 일이 발생할 수 있다는 것이다.



이 문제는 벽의 크기를 늘릴 때만 발생한다. (Why ?), 이 문제를 방지하려면 벽의 크기를 늘릴 때, 볼이 오른쪽으로 움직이면서 벽이 그려질 영역을 벗어나는 순간 그려야한다는 것이다. 따라서 벽의 크기가 커져야 할 때 볼의 움직임에 벽이 따라가는 모양이 만들어진다. (실제 이 과정을 부드럽게 하려면, 벽을 점진적으로 그려야 하지만, 한 단위의 블록을 한 번에 그리도록 한다)

실제 이 프로그램에는 초기에 Semaphore를 선언한 뒤, 벽을 그리는 쪽에서 문제가 생길 소지가 있는 시점-즉 Ball이 벽을 그려야 할 자리를 지나고 있다면)에 세마포를 Take (또는 Give)하고, Ball을 그리는 쪽에서 안전한 시점으로 벗어나는 순간 세마포를 Give (또는 Take) 하는 방법으로 동기화를 한다. 필요에 따라 세마포를 여러 개 쓸 수 도 있다.

또 다른 기능은 왕복 거리와 상관없이 Ball이 2초에 한 번씩 좌우로 움직이게 만들어야 한다는 것이다. (Ball_Task의 우선 순위가 상대적으로 낮기 때문에 매우 정확한 2초를 유지할 수는 없다.) 따라서 벽이 길 때는 Ball이 천천히 움직이고, 벽이 짧을수록 Ball이 빨리 움직이게 된다. (vTaskDelayUntil() 함수를 이용)

이 실험은 2 주간 작업을 한다.

5-1. 소스 및 샘플 프로그램

A. Template 소스는 자료실의 Ball-Wall.zip에 있음
B. 미리 만들어 놓은 Binary는 자료실의 Ball_Wall.nds 임

5-2. 소스 및 샘플 프로그램 (Github 이용)

소스는Github의 Ball 소스를 check-out하여 과제를 수행한다.

https://github.com/hl1itj/nds-ide/tree/microprocessor/lab/ball-wall

에서, Checkout하여 과제를 수행하고 (main.c, exp_homework.c를 필요한 만큼 수정하여 구현한다..)

(이 문서, 미리 build한 Ball-Wall,nds는 doc directory에 있음)

https://github.com/hl1itj/nds-ide/tree/microprocessor/학번/ball-wall

에서 각자 개발을 진행한다.